Lesson 11
Graphs and Graph Traversal

Combinatorics of Pure Intelligence

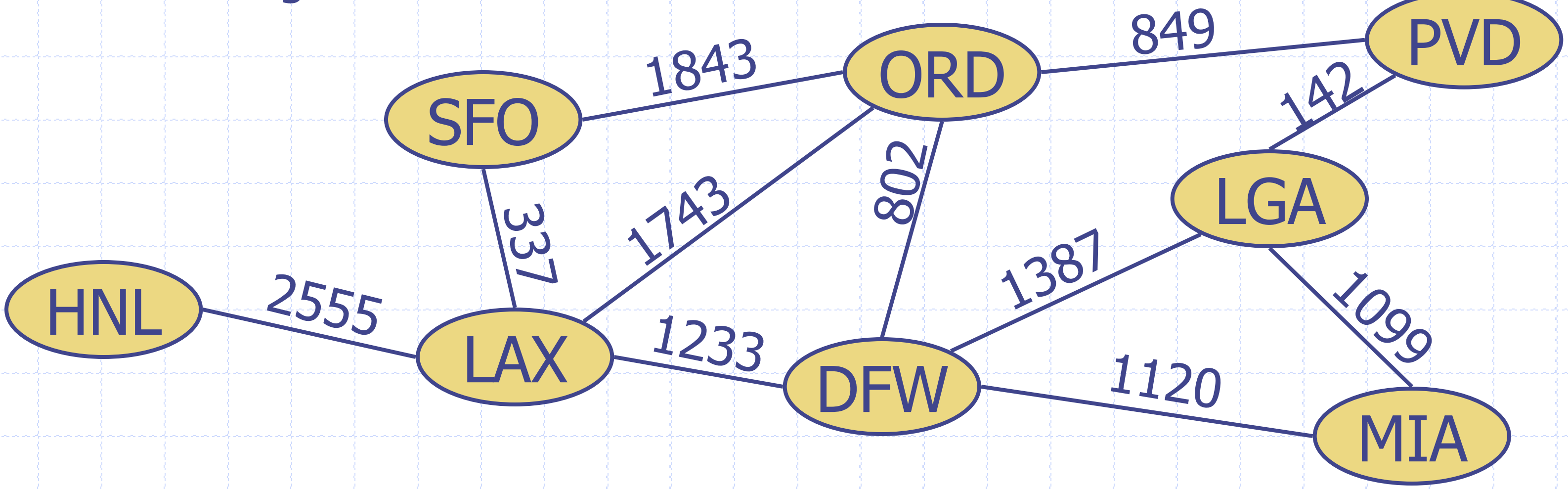
Wholeness of the Lesson

Graphs are data structures that do more than simply store and organize data; they are used to model interactions in the world. This makes it possible to make use of the extensive mathematical knowledge from the theory of graphs to solve problems abstractly, at the level of the model, resulting in a solution to real-world problems.

Science of Consciousness: Our own deeper levels of intelligence exhibit more of the characteristics of Nature's intelligence than our own surface level of thinking. Bringing awareness to these deeper levels, as the mind dives inward, engages Nature's intelligence, Nature's know-how, and this value is brought into daily activity. The benefit is greater ability to solve real-world problems, meet challenges, and find the right path for success.

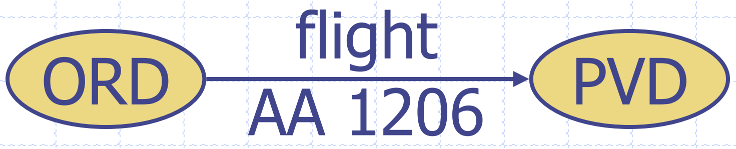
Graphs

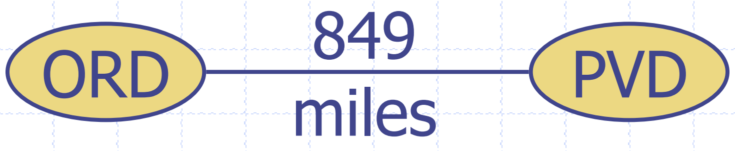
- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges can be implemented so that they store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
  - form directed graphs
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
  - form undirected graphs
- Weighted edge
  - given a weight
  - weight could represent cost, distance, etc.
  - form weighted graphs



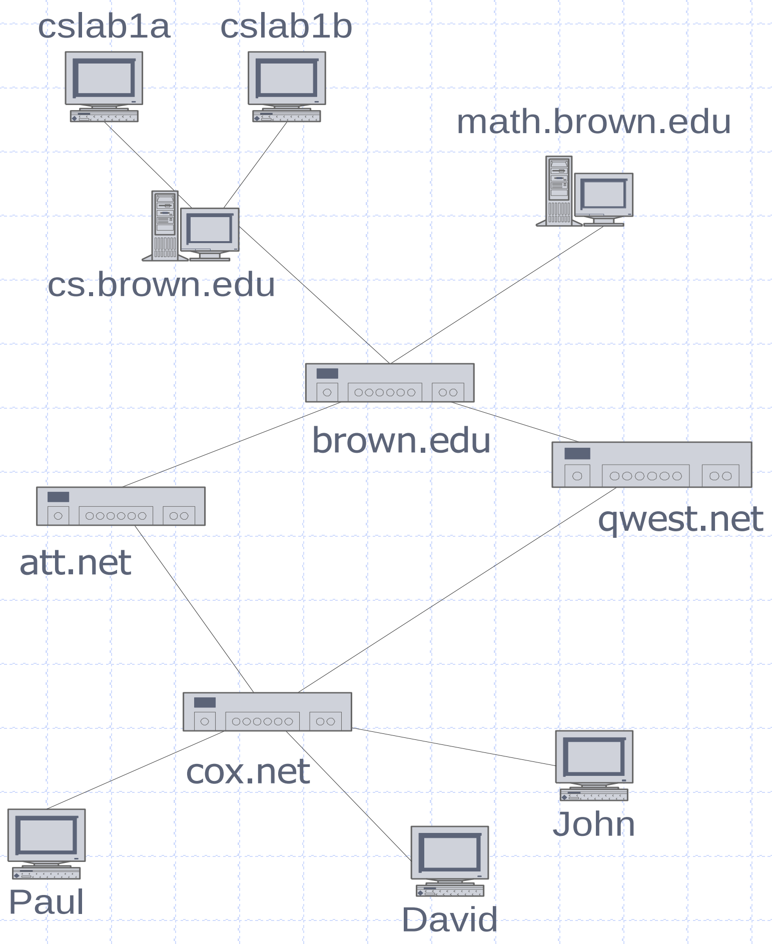


- Unweighted edge
  - no weight on it
  - form unweighted graphs

Applications

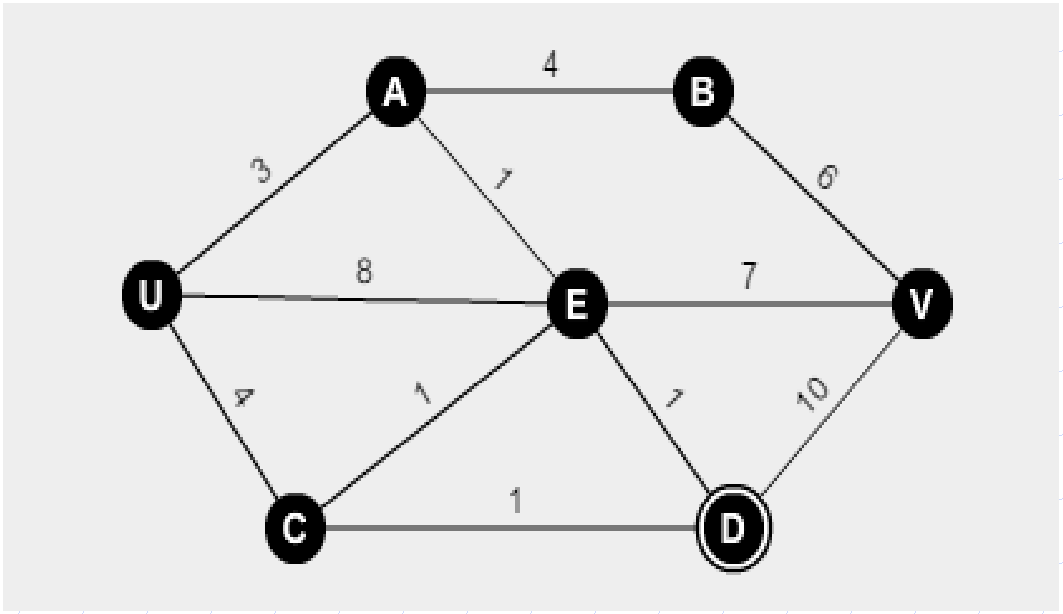
- Electronic circuits
  - Printed circuit board

    (nodes = junctions, edges are the traces)
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram
- Physics / Chemistry
  - Atomic structure simulations (e.g. shortest path algs)
  - Model of molecule -- atoms/bonds



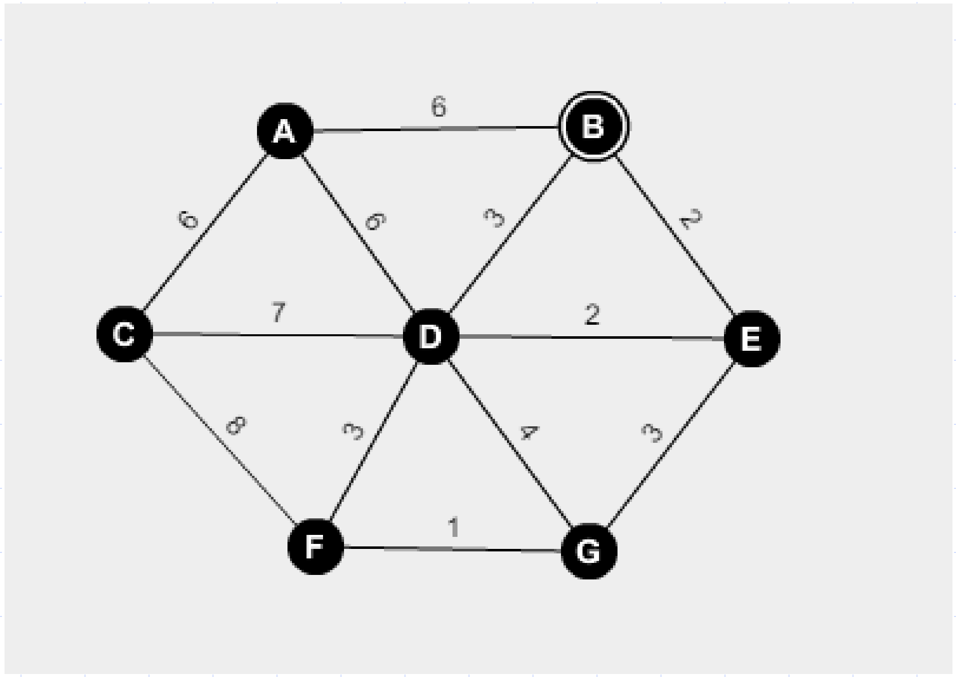
Examples

SHORTEST PATH. The diagram below schematically represents a railway network between cities; each numeric label represents the distance between respective cities. What is the shortest path from city U to city V? Devise an algorithm for solving such a problem in general.



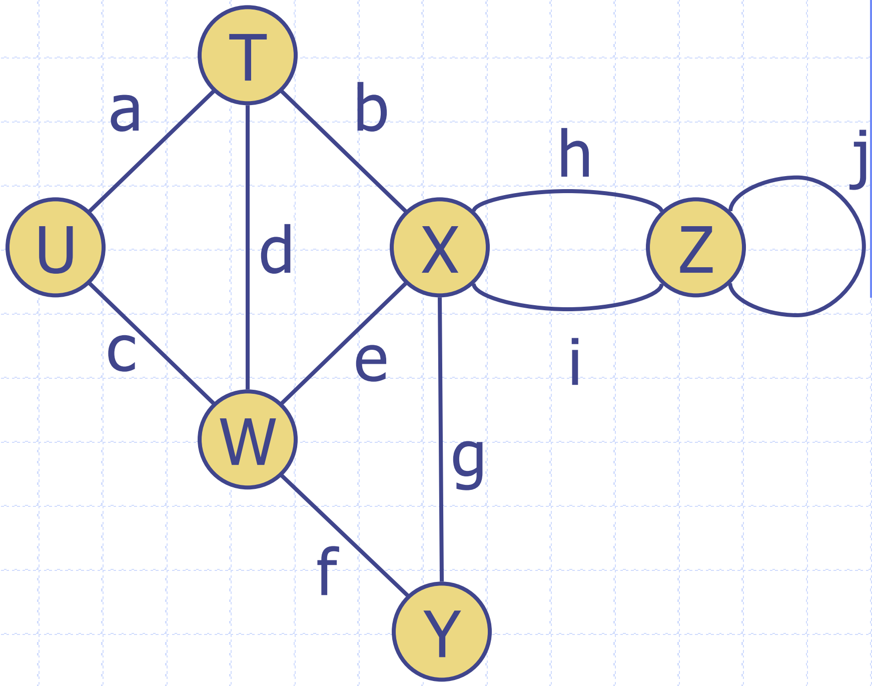
Examples (continued)

CONNECTOR. The diagram below schematically represents potential railway paths between cities; a numeric label represents the cost to lay the track between the respective cities. What is the least costly way to build the railway network in this case, given that it must be possible to reach any city from any other city by rail? Devise an algorithm for solving such a problem in general.



Terminology

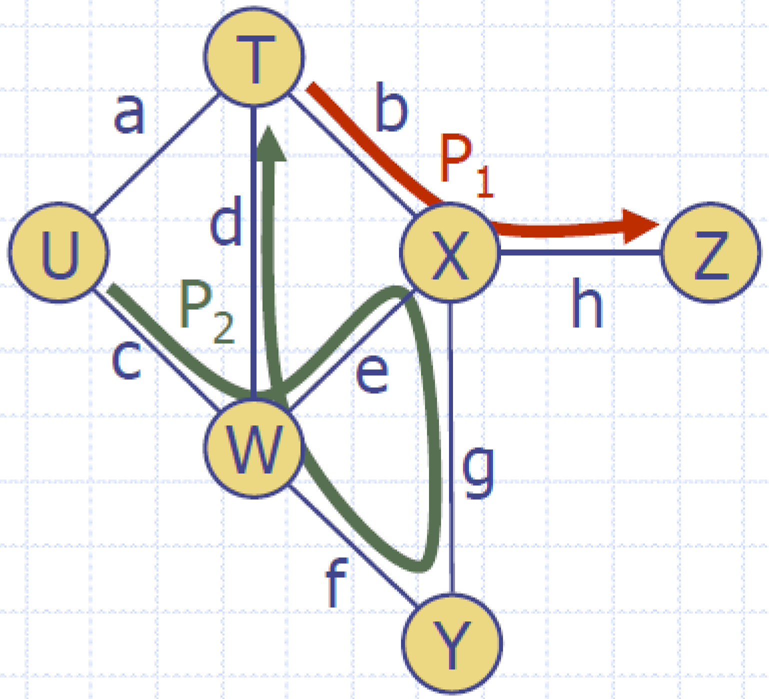
- |V| (or *n*) is the number of vertices of G; |E| (or *m*) is the number of edges.
- End vertices (or endpoints) of an edge
  - U and T are the endpoints of a
- Edges incident to a vertex
  - a, d, and b are incident to T
- Adjacent vertices
  - U and T are adjacent
- Degree of a vertex: number of edges incident to it
  - X has degree 5 denoted as: deg(X) = 5



- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop
- Simple Graph
  - A simple graph is a graph that has no self-loops or parallel edges

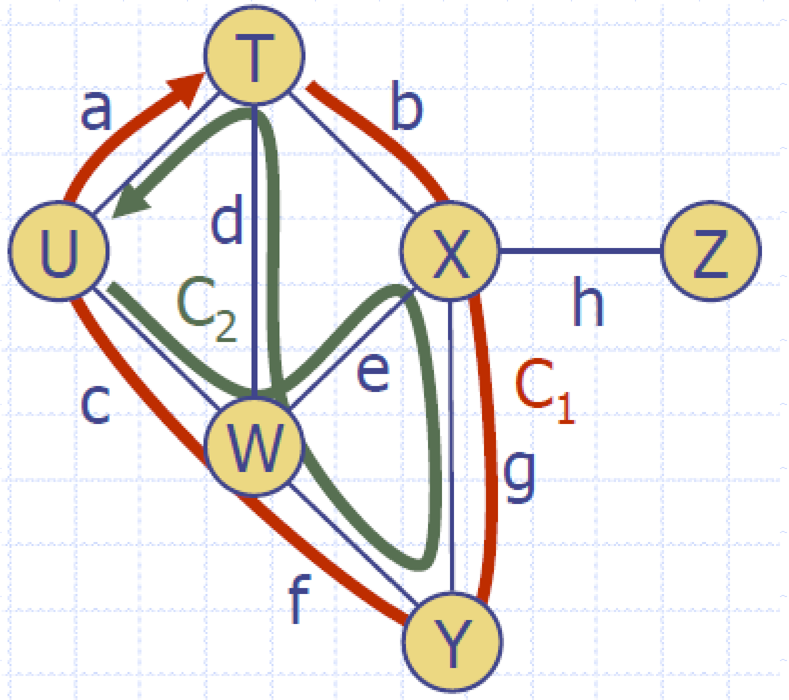
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges - in a simple graph, can omit edges
  - begins and ends with a vertex
  - length of a path is number of edges
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(T, X, Z) is a simple path
  - P_2=(U, W, X, Y, W, T) is a path that is not simple



Terminology (cont.)

- Cycle
  - path in which all edges are distinct and whose first and last vertex are the same
  - *length* of a cycle is the number of edges in the cycle
- Simple cycle
  - path such that all vertices and edges are distinct except first and last vertex
- Examples
  - C_1=(T, X, Y, W, U, T) is a simple cycle
  - C_2=(U, W, X, Y, W, T, U) is a cycle that is not simple



Properties

Convention: Focus on simple undirected graphs at first

Property 1

$$\Sigma_{\mathbf{v}} \deg(\mathbf{v}) = 2m$$

Proof: Let E_v = {e | e incident to v}.
Then

$$\Sigma_{\mathbf{v}} \deg(\mathbf{v}) = \Sigma_{\mathbf{v}} |E_{\mathbf{v}}|$$

Notice every edge (v,w) belongs to just two of these sets: E_v and E_w. So every edge is counted exactly twice.

Property 2

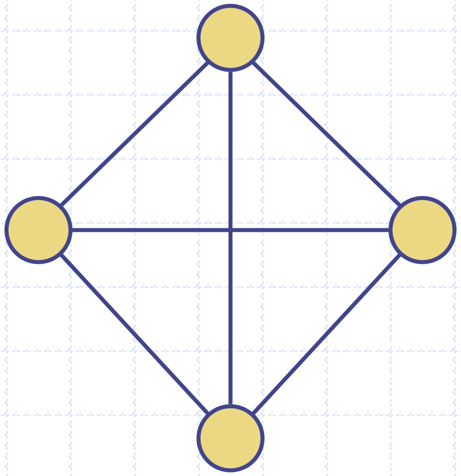
$$m \le n(n-1)/2$$

Proof: max number of edges is

$$C(n, 2) = C_{n,2} = n(n-1)/2$$

Notation

n	number of vertices
m	number of edges
$\deg(\mathbf{v})$	degree of vertex $\mathbf{v}$

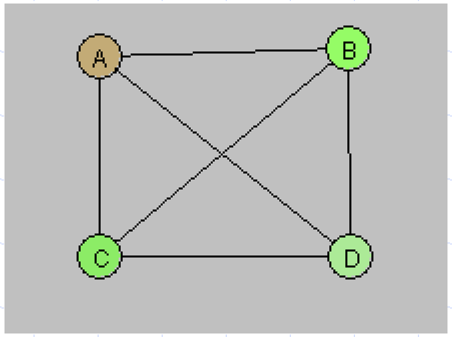


Example

- $n = 4$
- $m = 6$
- $\deg(\mathbf{v}) = 3$

Complete Graphs

- A graph G is complete if for every pair of vertices (u,v), there is an edge (u,v) in G.
- This is the complete graph on 4 vertices, denoted K_4.



- In general, the complete graph on n vertices is denoted K_n.
- <u>Fact.</u> For a complete graph G,

  $m = n(n-1)/2$

  that is to say, K_n has exactly n(n-1)/2 edges.

Subgraphs

A graph $H = (V_H, E_H)$ is a **subgraph** of a graph $G = (V_G, E_G)$ if both of the following are true:

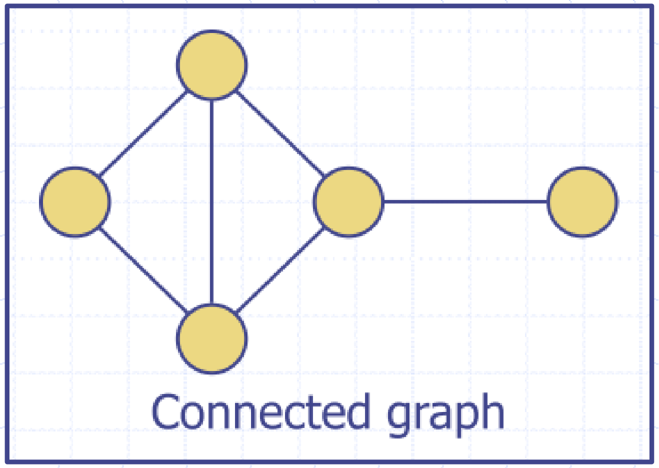
a. $V_H \subseteq V_G$ and $E_H \subseteq E_G$, and

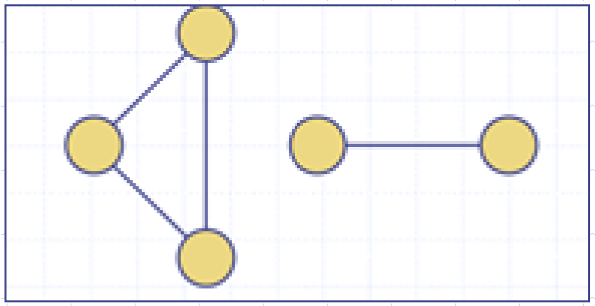
b. for every edge (u, v) belonging to E_H, both u and v belong to V_H.

H is called a **spanning subgraph** if $V_H = V_G$.

Connected Graphs And Connected Components

- A graph is connected if for any two vertices u,v in G, there is a path from u to v.
- **Observation:** If a graph G=(V,E) is not connected, then G can be partitioned into different components. Each component is connected and is connected to no additional vertices in the supergraph. They are called the connected components of G.

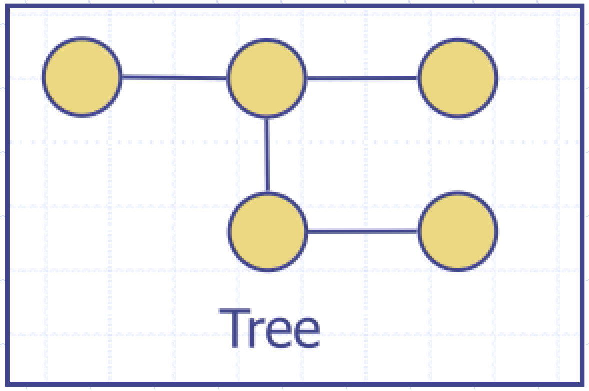




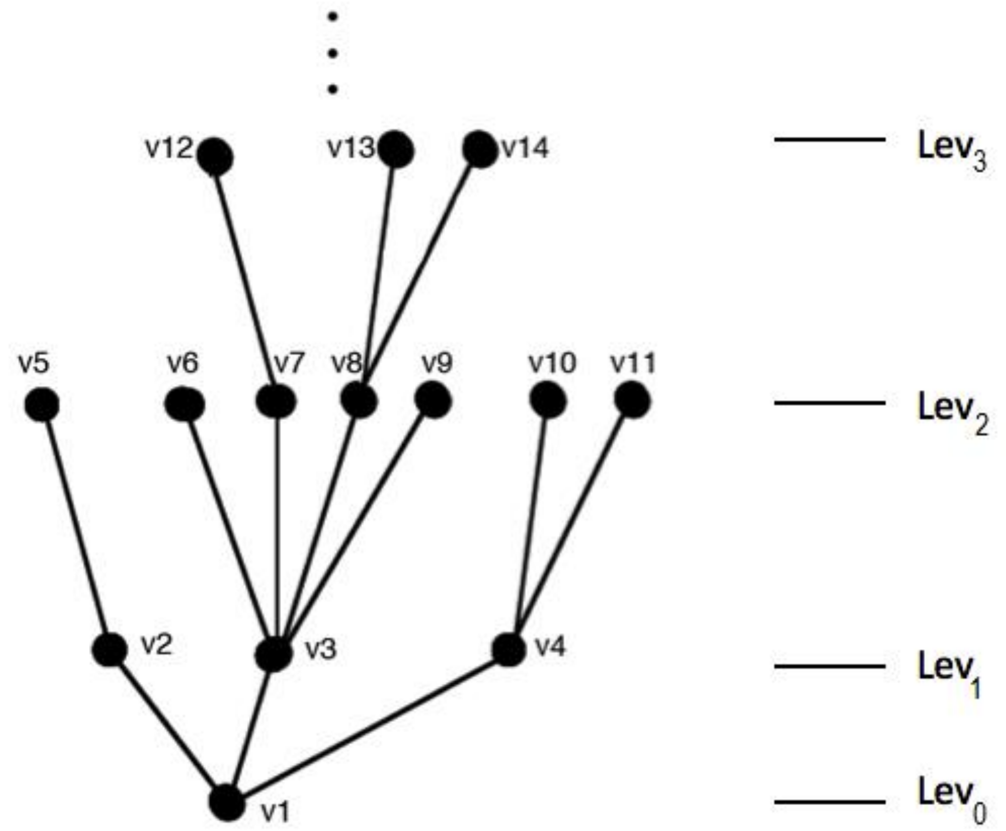
Disconnected graph with two connected components

Trees

- A graph is **acyclic** if it contains no cycle.
- An acyclic connected graph is called a **tree**.
- A **rooted tree** is a tree having a distinguished vertex r.
  - The usual levels we have in trees as data structures can be defined for rooted trees. Level 0 contains only the root. Level 1 contains all vertices adjacent to the root. Level 2 contains all vertices adjacent to a Level 1 vertex other than the Level 0 vertex. In general, Level i + 1 contains vertices adjacent to the Level i vertices that do not appear at previous levels.



A Rooted Tree with Levels Shown

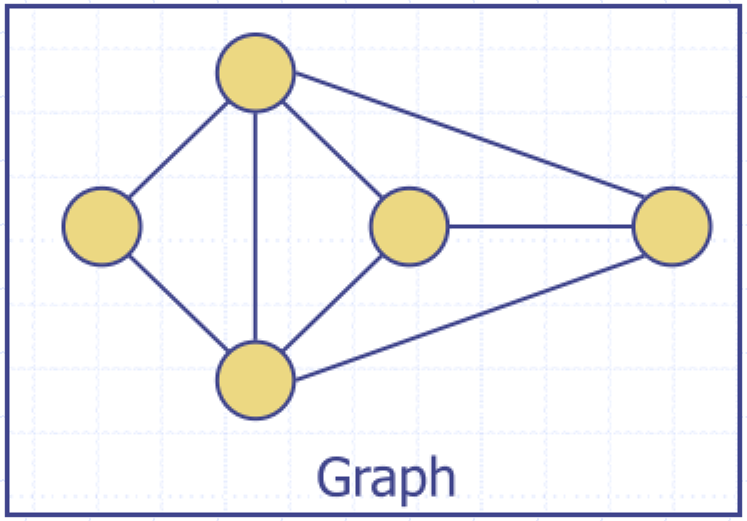


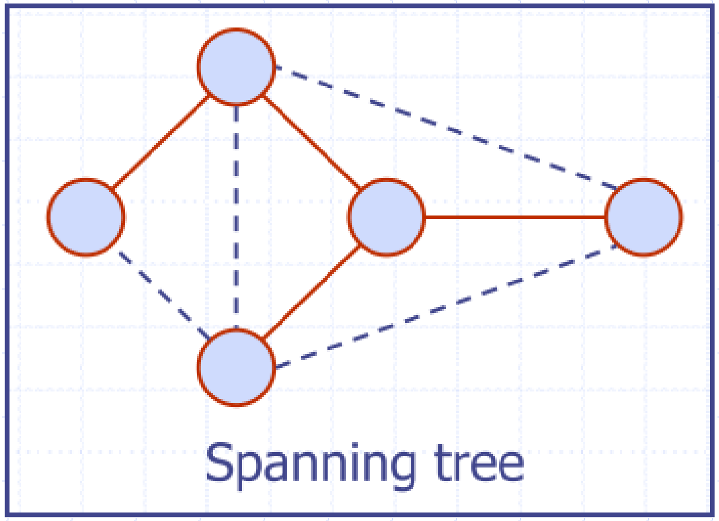
Some Theorems About Trees

- **Theorem**. In a tree, any two vertices are connected by a unique simple path.
- **Theorem**. If G is a tree, $m = n - 1$.

Spanning Trees and Forests

- A ***spanning tree*** of a graph is a subgraph of the original graph which is a tree that contains all vertices of the original graph.
- A spanning tree is not unique unless the graph is itself a tree.
- If the graph is not connected, a spanning forest will be obtained by putting together spanning trees from each connected component.





Hamiltonian Graphs And Vertex Covers

- A **Hamiltonian cycle** in a graph G is a simple cycle that contains every vertex of G. A graph is a **Hamiltonian graph** if it contains a Hamiltonian cycle.
- If G = (V,E) is a graph, a **vertex cover for** G is a set C $\subseteq$ V such that for every e $\in$ E, at least one end of e lies in C.
- **Fact**. The known algorithms for determining whether a graph is Hamiltonian, and for computing the smallest size of a vertex cover, run in exponential time.

Main Point

The body of knowledge in the field of Graph Theory becomes accessible in a practical way through the Graph Abstract Data Type, which specifies the computations that a Graph software object should support. The Graph Data Type is analogous to the human physiology: The abstract intelligence that underlies life relies on the concrete physiology to find expression in thinking, feeling, and behavior in the physical world.

Operations on Graphs and Efficient Implementation

- Next we will focus on a useful set of operations on a graph and efficient implementation. Efficiency here will depend both on the algorithm and the implementation.
- *The Graph ADT.* There is no standard set of operations for a graph. We emphasize several that are useful in applications of graphs.

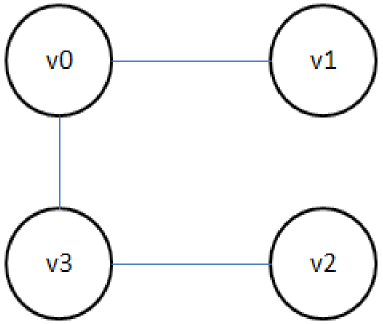
```
boolean areAdjacent(Vertex u, Vertex v)
List getListOfAdjacentVerts (Vertex u)
Graph getSpanningTree()
List getConnectedComponents()
boolean isConnected()
boolean hasPathBetween(Vertex u, Vertex v)
boolean containsCycle()
boolean isTree()
int lengthOfShortestPath(Vertex u, Vertex v)
```

- Demo: The Graph class.

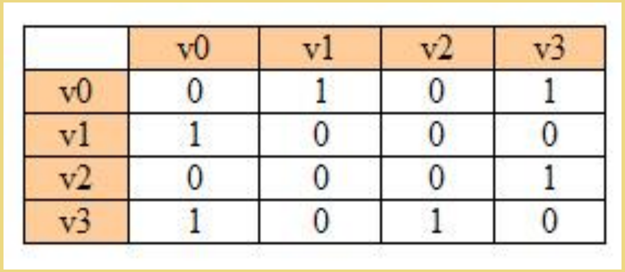
Determining Adjacency

- Most graph algorithms rely heavily on determining whether two vertices are adjacent and on finding all vertices adjacent to a given vertex. These operations should be as efficient as possible.
- *Two ways to represent adjacency.*
  - Adjacency Matrix
  - Adjacency List

Determining Adjacency

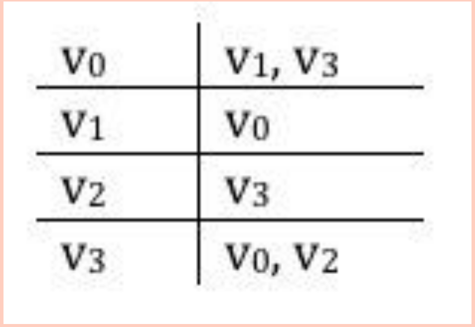


- An *Adjacency Matrix A* is a two-dimensional n x n array consisting of 1's and 0's. A 1 at position *A*[i][j] means that vertices v_i and v_j are adjacent; 0 indicates that the vertices are not adjacent.



	v0	v1	v2	v3
v0	0	1	0	1
v1	1	0	0	0
v2	0	0	0	1
v3	1	0	1	0

- An *Adjacency List* is a table that associates to each vertex *u* the set of all vertices in the graph that are adjacent to *u.*



V_0	V_1, V_3
V_1	V_0
V_2	V_3
V_3	V_0, V_2

Determining Adjacency

- If there are relatively few edges, an adjacency matrix uses too much space. Best to use adjacency list when number of edges is relatively small.
- If there are many edges, determining whether two vertices are adjacent becomes costly for an adjacency list. Best to use adjacency matrix when there are many edges.
- Demo: Adjacency in the Graph class.

Sparse Graphs vs Dense Graphs

- Recall the maximum number of edges in a graph is n(n - 1)/2, where n is the number of vertices.
- A graph is said to be **dense** if it has $\Theta(n^2)$ edges. It is said to be **sparse** if it has O(n) edges.
- **Strategy**. Use adjacency lists for sparse graphs and adjacency matrices for dense graphs.
- For purposes of implementation, in this class we will use adjacency lists. But for purposes of determining optimal running time, we will assume we are using whichever implementation gives the best running time.

Graph Traversal Algorithms

- **Motivation.** Graph traversal algorithms provide efficient procedures for visiting every vertex in a graph. There are many practical reasons for doing this.
  - Telephone network – check whether there is a break in the network
  - Driving directions – some graph traversal algorithms tell you the shortest path from one vertex to another

Depth-First Search

- Depth-first search is an example of a graph traversal algorithm. At each vertex, it is possible to do additional processing, but the basic algorithm just shows how to visit each vertex in the graph.
- The basic DFS strategy: Pick a starting vertex and visit an adjacent vertex, and then one adjacent to that one, and so on, until a vertex is reached that has no further unvisited adjacent vertices. Then backtrack to the mostly recently visited one that does have an unvisited adjacent vertex, and follow that path. Continue in this way till all vertices have been visited.

Depth-First Search

Algorithm: Depth First Search (DFS)

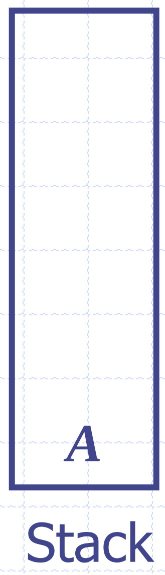
Input: A simple connected undirected graph G = (V,E)

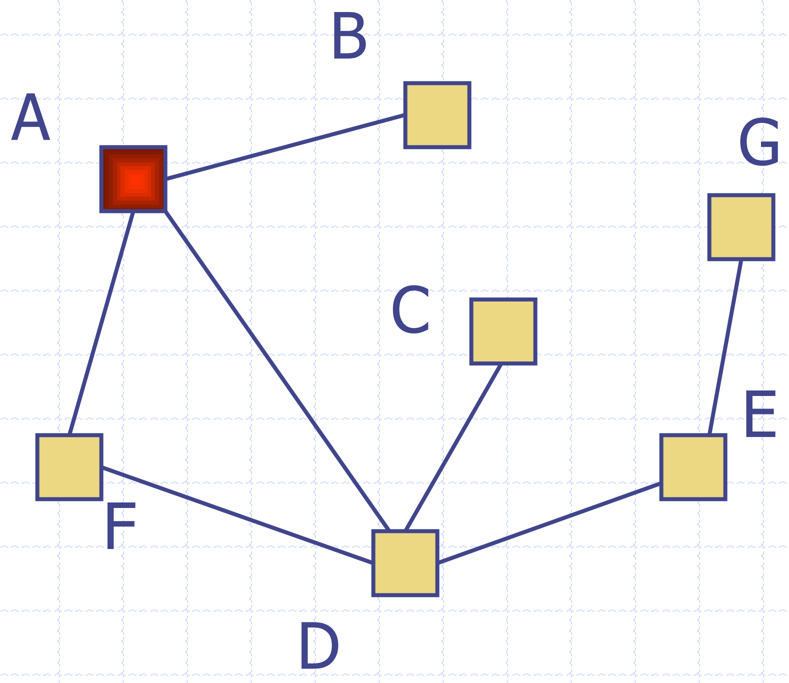
Output: G, with all vertices marked as visited.

```
Initialize a stack S   //supports backtracking
Pick a starting vertex s and mark it as visited
S.push(s)
while S ≠ ∅ do
    v ← S.peek()
    if some vertex adjacent to v not yet visited then
        w ← next unvisited vertex adjacent to v
        mark w
        push w onto S
    else //if can't find such a w, backtrack
        S.pop()
```

Worked Example

Start with Vertex A, mark as visited and push into stack.

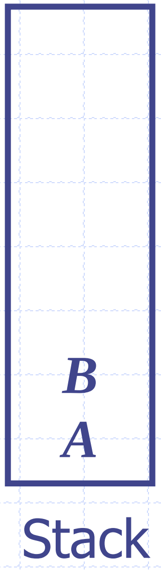


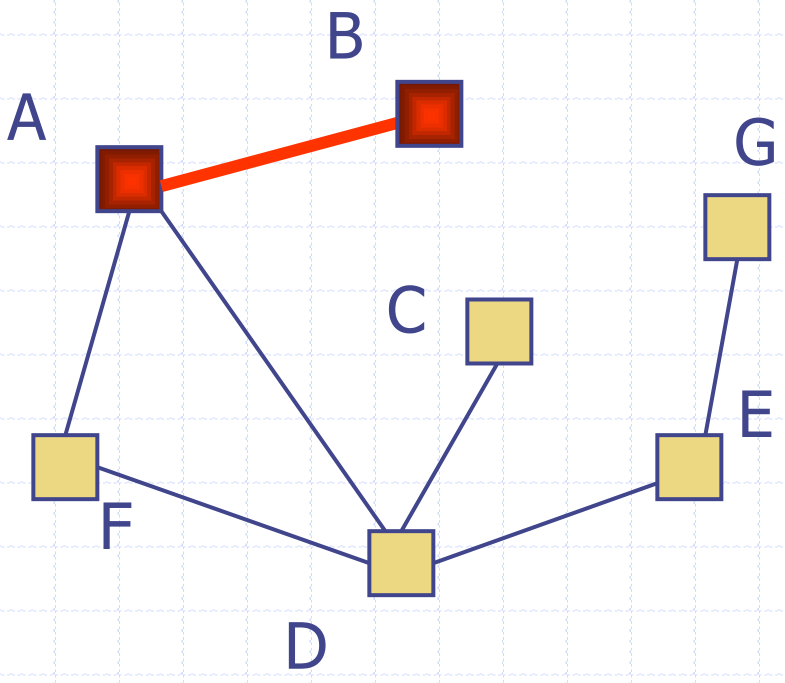


Spanning Tree:
T = {…}

Worked Example

Peek at the stack and find A; B is unvisited and adjacent to A; mark B as visited and push onto stack.

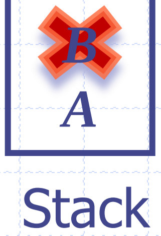


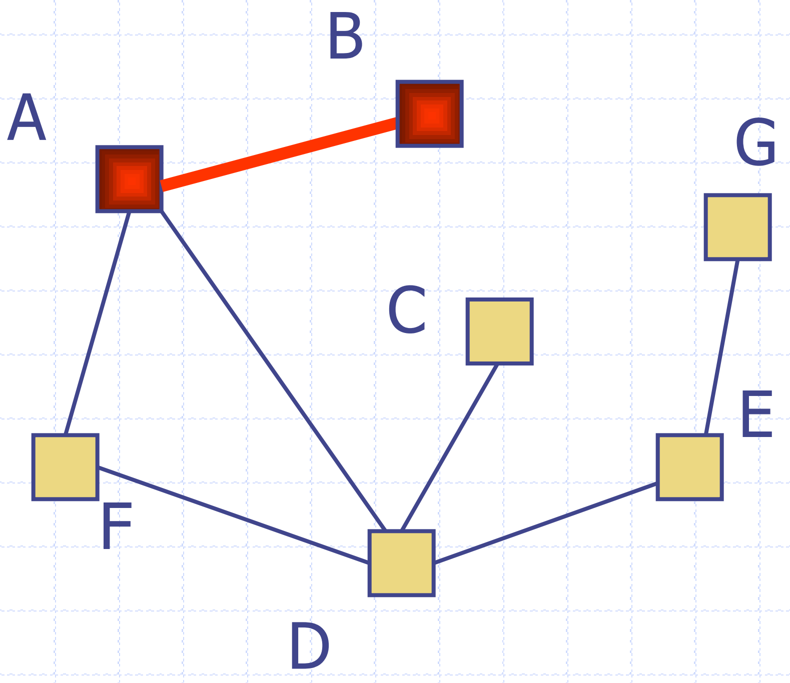


Spanning Tree:
T = {AB, …}

Worked Example

Peek at the stack and find B; B has no more unvisited vertices; pop the stack to remove B

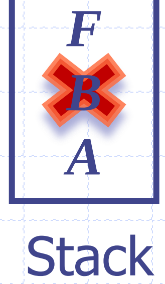


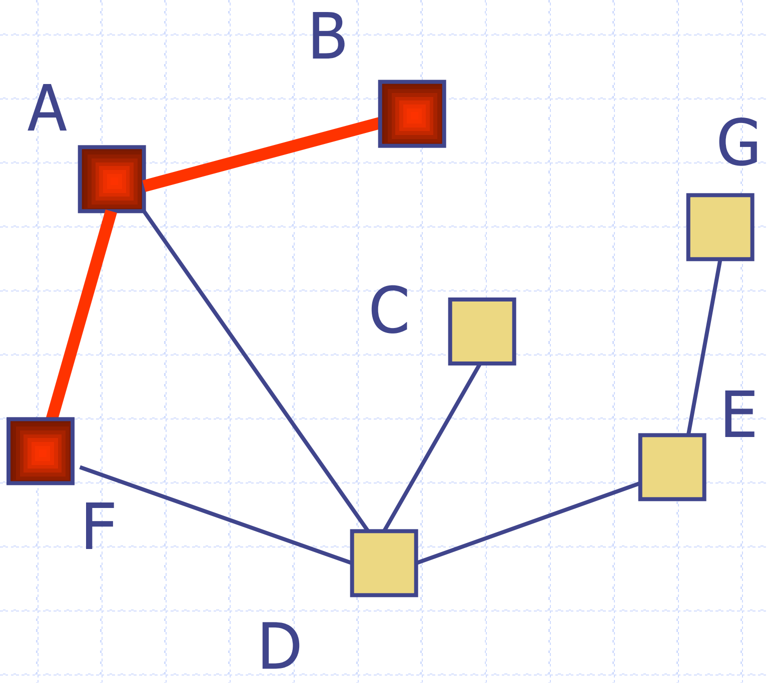


Spanning Tree:
T = {AB, …}

Worked Example

Peek at the stack and find A; F is adjacent to A and not yet visited; mark F as visited and push it into the stack

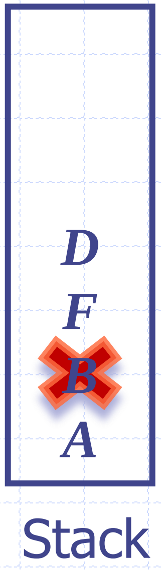


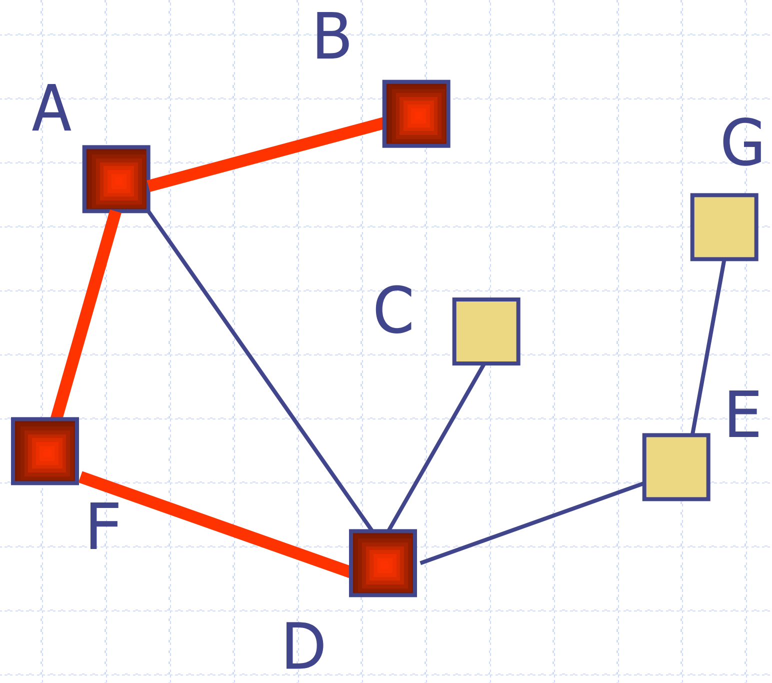


Spanning Tree: T = {AB, AF, …}

Worked Example

Peek at the stack and find F; D is adjacent to F and not yet visited; mark D as visited and push it into the stack

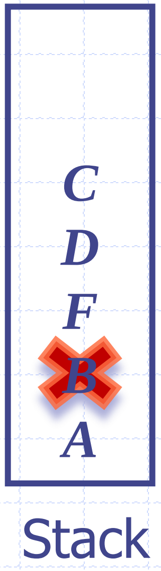


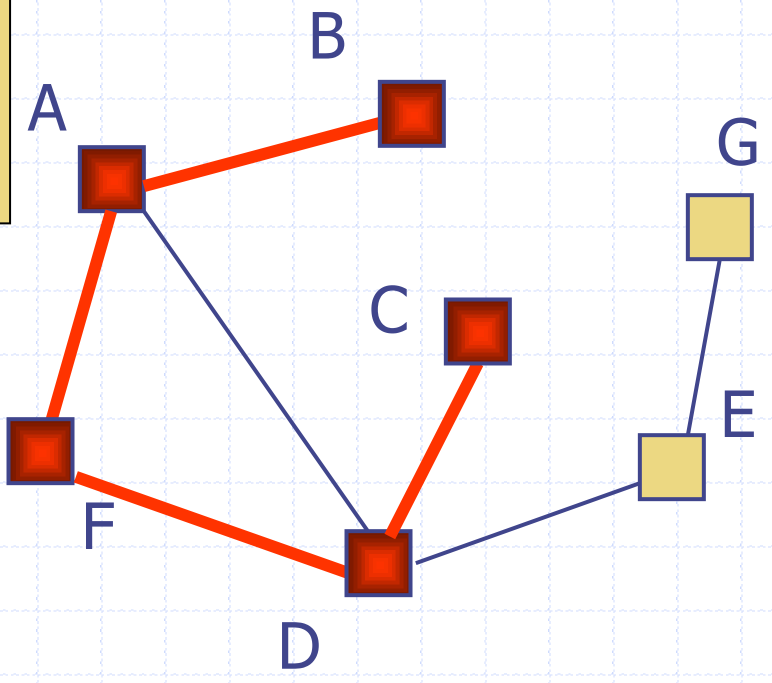


Spanning Tree: T = {AB, AF, FD, …}

Worked Example

Peek at the stack and find D; we do not go back to A since A was visited (avoids cycle creation); C is adjacent to D and not yet visited, so mark C and push it.

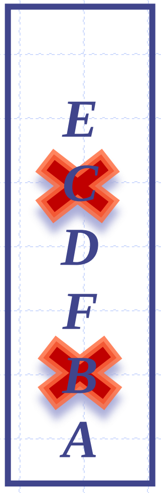




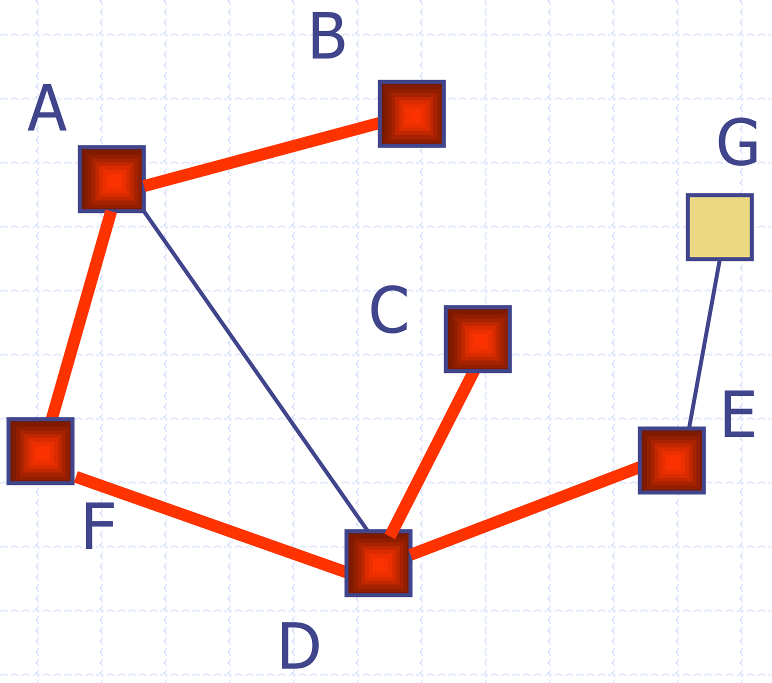
Spanning Tree: T = {AB, AF, FD, DC…}

Worked Example

Peek at the stack and find C; C has no more unvisited vertices, so pop C. Peek at the stack and find D; E is adjacent to D and not yet visited; mark E and push onto the stack



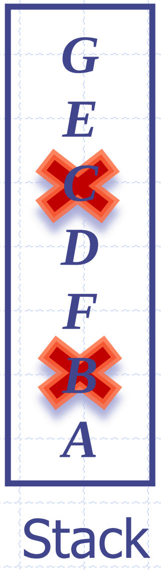
Stack

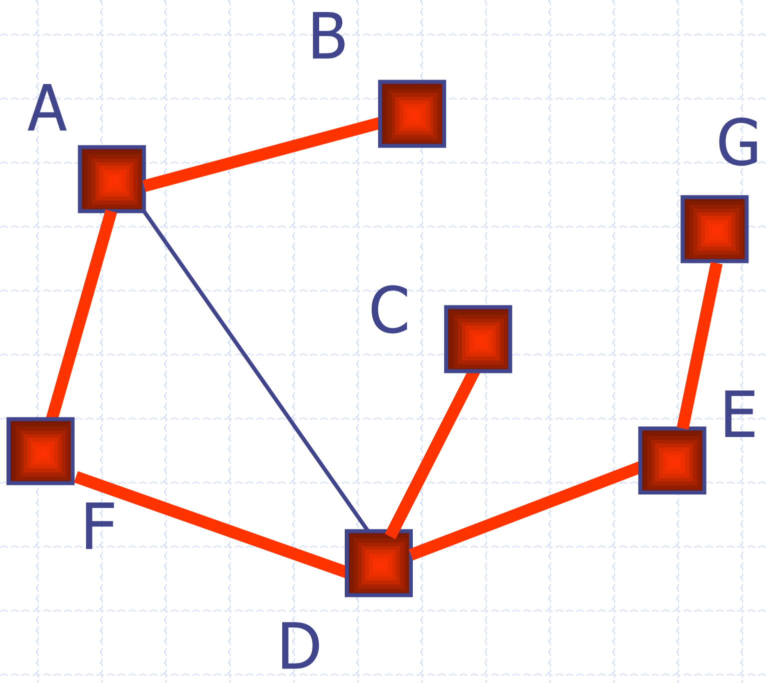


Spanning Tree: T = {AB, AF, FD, DC, DE…}

Worked Example

Peek at the stack and find E; G is adjacent to E and not yet visited; mark G and push onto the stack

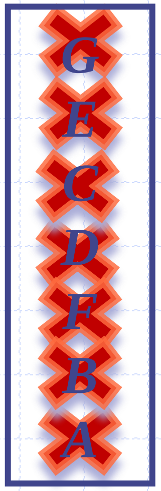




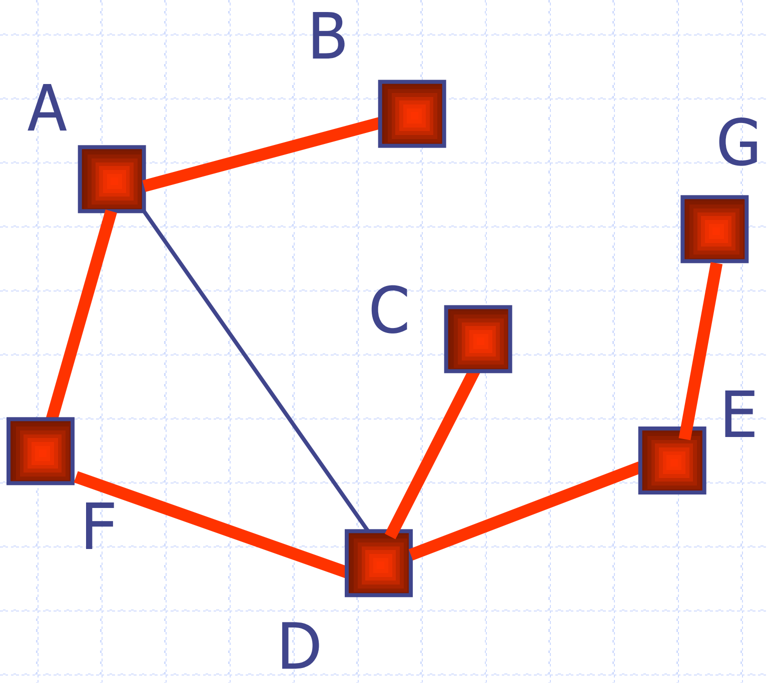
Spanning Tree: T = {AB, AF, FD, DC, DE, EG…}

Worked Example

Now we have no more unvisited vertices, so pop all vertices that are still in the stack.



Stack



Spanning Tree: T = {AB, AF, FD, DC, DE, EG}

Some Implementation Issues - how to indicate a vertex has been visited

- Need to decide how to represent the notion that a vertex "has been visited". Often this is done by creating a special bit field for this purpose in the Vertex class.
- An alternative is to view visiting as being conducted by DFS, so DFS is responsible for tracking visited vertices. Can do this with a hashtable – insert (u,u) whenever u has been visited. Checking whether a vertex has been visited can then be done in O(1) time. (We use this approach.)

Some Implementation Issues - Code for Efficient Access of Adjacency List

```
public Vertex nextUnvisitedAdjacent(Vertex v) {
        List<Vertex> listOfAdjacent = adjacencyList.get(v);
        Iterator<Vertex> it = listOfAdjacent.iterator();
        Vertex retVert = null;
        while(it.hasNext()) {
            Vertex u = it.next();
                if(visitedVertices.containsKey(u)) {
                    it.remove();
                }
                if(!visitedVertices.containsKey(u)) {
                    retVert = u;
                    it.remove();
                    return retVert;
                }
        }
        return retVert;
}
```

Some Implementation Issues – Copy of adjacency list

- The adjacency list that is used during DFS will be only a copy of the original.

```
public HashMap<Vertex,LinkedList<Vertex>> getAdjacencyList() {

    HashMap<Vertex,LinkedList<Vertex>> copy
        = new HashMap<Vertex,LinkedList<Vertex>>();
    for(Vertex v : adjList.keySet()) {
            copy.put(v, getListOfAdjacentVerts(v));

    }
    return copy;
}
```

Running Time for DFS

- Every vertex eventually is marked and pushed onto the stack, and then is eventually popped from the stack, and each of these occurs only once. Therefore, each vertex undergoes O(1) steps of processing.
- In addition, each vertex v will experience a peek operation, at which time the algorithm will search for an unvisited vertex adjacent to v. This peek step, together with the search, will take place repeatedly until every vertex adjacent to v has been visited – in other words, deg(v) times.
- Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum over all v in V is

$$\mathbf{O(\Sigma_v\,(1 + deg(v))) = O(n + 2m) = O(n+m)}$$

DFS for Disconnected Graphs

- Question: How to make changes to the following algorithm so it will work for disconnected graphs?

```
Algorithm: Depth First Search (DFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.
    Initialize a stack S   //supports backtracking
    Pick a starting vertex s and mark it as visited
    S.push(s)
    while S ≠ ∅ do
        v ← S.peek()
        if some vertex adjacent to v not yet visited then
            w ← next unvisited vertex adjacent to v
            mark w
            push w onto S
        else //if can't find such a w, backtrack
            S.pop()
```

DFS for Disconnected Graphs

```
Algorithm: Depth First Search (DFS)
Input: A simple undirected graph G = (V,E)
Output: G, with all vertices marked as visited.
  while there are vertices unvisited do
      s ← next unvisited vertex
      mark s as visited
      initialize a stack S
      S.push(s)
      while S ≠ ∅ do
          v ← S.peek()
          if some vertex adjacent to v not yet visited then
              w ← next unvisited vertex adjacent to v
              mark w
              push w onto S
          else
              S.pop()
```

Finding a Spanning Tree (Forest) with DFS

- A spanning tree for a connected graph can be found by using DFS and recording each new edge as it is discovered. If the graph is not connected, the algorithm can be done in each component to create a *spanning forest.*

- *Running Time.* The extra steps added to keep track of discovered edges is constant. Therefore, like DFS, running time is O(n + m).

Finding a Spanning Tree – change on top of DFS

```
Algorithm: Finding a Spanning Tree
Input: A simple connected undirected graph G = (V,E)
Output: a spanning tree T for G
    Initialize a stack S   //supports backtracking
    Initialize a List T
    Pick a starting vertex s and mark it as visited
    S.push(s)
    while S ≠ ∅ do
        v ← S.peek()
        if some vertex adjacent to v not yet visited then
            w ← next unvisited vertex adjacent to v
            add edge (v, w) to T
            mark w
            push w onto S
        else //if can't find such a w, backtrack
            S.pop()
    return T
```

Template Design Pattern

- Observation: Finding Spanning Tree is built on top of DFS, and most logic is same as DFS. Obviously, we don't want to repeat the same code during implementation.
- **Solution:** We can represent DFS as a class. Then, other algorithms (like FST) that make use of the DFS strategy can be represented as subclasses. To this end, insert "process" options at various breaks in the code to allow subclasses to perform necessary processing of vertices and/or edges. (This is an application of the *template design pattern.*)

AbstractGraphSearch

- **It is an abstract class – served as a template.**

```
Algorithm: AbstractGraphSearch
    while (some vertex not yet visited)
        handleInitialVertext()
        singleComponentLoop()
        additionalProcessing()   //by default it is doing nothing, we
                                 //will see how to make use of it later.
```

Rework DFS using the template

- Make DFS be a subclass of AbstractGraphSearch, then we need to take care of the abstract methods from AGS.

```
Algorithm: handleInitialVertex
  Initialize a stack S   //supports backtracking
  Pick a vertex s not yet visited
        S.push(s)
        mark s as visited
        processVertex(s)
```

Rework DFS using the template

```
**Algorithm**: SingleComponentLoop
    while $S \neq \emptyset$ do
        $v \leftarrow$ S.peek()
        if some vertex adjacent to v not yet visited then
            $w \leftarrow$ next unvisited vertex adjacent to v
            mark w as visited
            push w onto S
            processEdge(v, w)
            processVertex(w)
        else **//if can't find such a w, backtrack**
            S.pop()
```

These two methods again become template methods.

Rework Finding a Spanning Tree with the template

- We can make FST a subclass of DFS, then the only thing we need to change is the processEdge step.
- We can maintain a List<Edge> inside FST, then whenever we discover an edge in DFS, add it to the list.

```
Algorithm: processEdge(v, w)
        list.add(new Edge(v,w))
```

- Demo: AbstractGraphSearch, DepthFirstSearch and FindSpanningTree classes

Finding Connected Components

- Each time DFS completes a round, it completes a search on a single connected component. Therefore, we can track the rounds of DFS and assign numbers to vertices according to the round in which they are discovered. Round 0 vertices belong to component #0, round 1 vertices to component #1, etc.
- *Implementation.* Create another subclass of the DFS class to handle connected components. During the in-between rounds, increment the counter. Record the component number for each vertex using a hashtable.
- When DFS has completed, the vertices and edges can be organized into graphs and a list of graphs can be returned.

Finding Connected Components

- In practice, it is helpful to insert in the Graph object not only the list of connected components, but also the hashtable that indicates which component each vertex belongs to.

List of Components	
0	v_1, v_2, v_3, v_4
1	v_5, v_7, v_9
2	v_6, v_8, v_{11}
3	v_{10}

Component-Map

Vertex -> Component Number Map	
v_1	0
v_2	0
v_3	0
v_4	0
v_5	1
v_6	2
. . .	. . .

Vertex-Component-Map

Answering Questions About Connectedness and Cycles

- Once we have obtained the connected components, we can provide answers to questions that may be asked of the graph:
  - Is it connected?
  - Is there a path from given vertices u and v?
  - Does the graph contain a cycle?

Answering Questions About Connectedness and Cycles

- Once we have obtained the connected components, we can provide answers to questions that may be asked of the graph:
  - Is it connected?
    - Answer: The graph is connected if and only if the number of connected components is 1. (Running time: O(1), once connected components are known.)
  - Is there a path from given vertices u and v?
    - Answer: There is a path from u to v if and only if u and v belong to the same component. (Running time: O(1), once connected components are known.)
  - Does the graph contain a cycle?

Answering Questions About Connectedness and Cycles

- *Cycle.* Each component is connected. Suppose the components are C_0, C_1, C_2,…, C_k. For each i, let n_i be the number of vertices and m_i the number of edges. If for some i, m_i is not equal to n_i – 1, then this component must contain a cycle: If not, then this component is connected and acyclic and therefore a tree, and so it must satisfy $m_i = n_i$ - 1. Contradiction!

- Therefore, the criterion for existence of a cycle is:

  G has a cycle if and only if there is a connected component C_i of G such that the number n_i of vertices of C_i satisfies $m_i \neq n_i - 1$, where m_i is the number of edges in C_i.

Main Point

To answer questions about the structure of a graph *G*, such as whether it is connected, whether there is a path between two given vertices, and whether the graph contains a cycle, it is sufficient to use DFS to compute the connected components of the graph. Based on this one piece of information, all such questions can be answered efficiently. This phenomenon illustrates the SCI principle of the *Highest First*: Experience the home of all knowledge first, and all particular expressions of knowledge become easily accessible.

Breadth-First Search

- An equally efficient way to traverse the vertices of a graph is to use the *Breadth First Search* (BFS) algorithm.
- *Idea.* Pick a starting vertex. Visit every adjacent vertex. Then take each of those vertices in turn and visit every one of its adjacent vertices. And so forth.

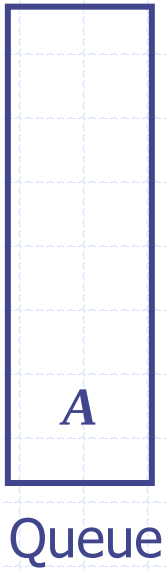
Breadth-First Search

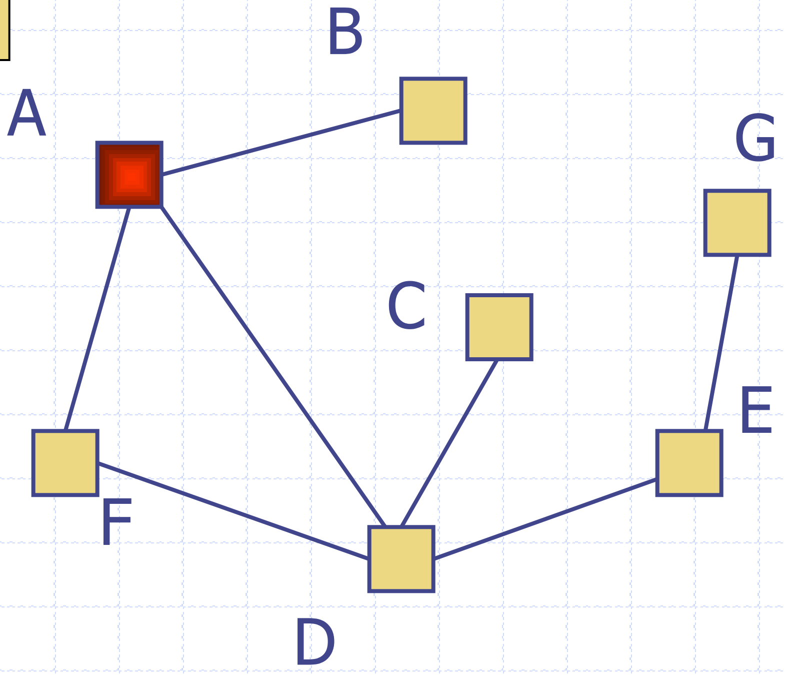
Algorithm: Breadth First Search (BFS)
Input: A simple connected undirected graph $G = (V,E)$
Output: G, with all vertices marked as visited.

```
Initialize a queue Q
Pick a starting vertex s and mark s as visited
Q.add(s)
while $Q \neq \emptyset$ do
    v $\leftarrow$ Q.dequeue()
    for each unvisited w adjacent to v do
        mark w
        Q.add(w)
```

Worked Example

Start with A, mark it as visited and add it to queue.

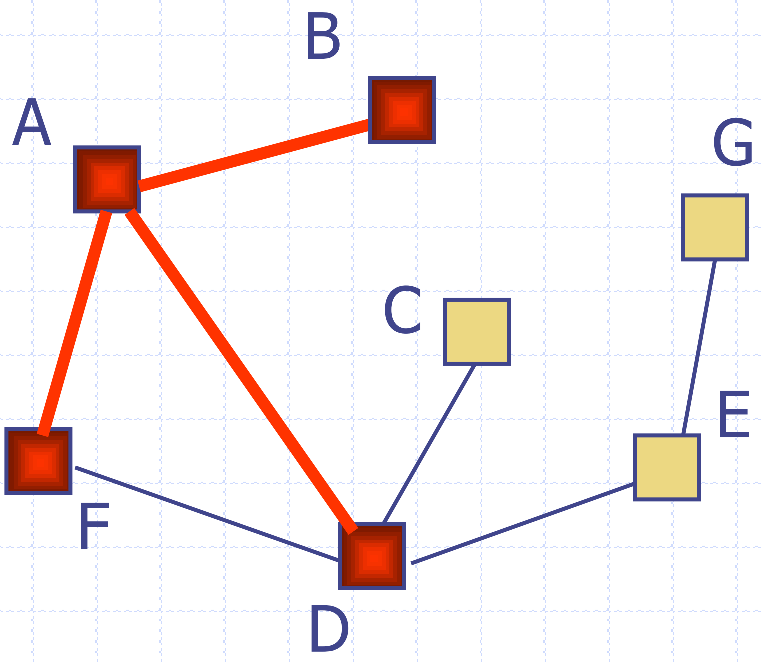


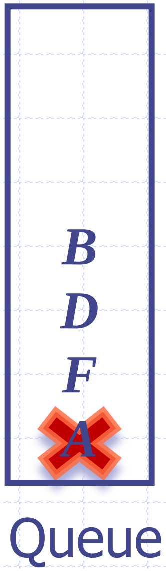


Spanning Tree: T = {…}

Worked Example

Dequeue (removes A); B, D and F are adjacent to A and not yet visited, so mark all three and add them to queue.

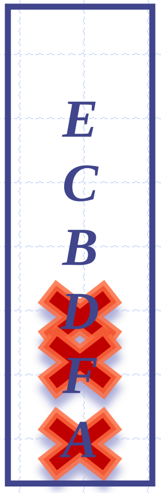




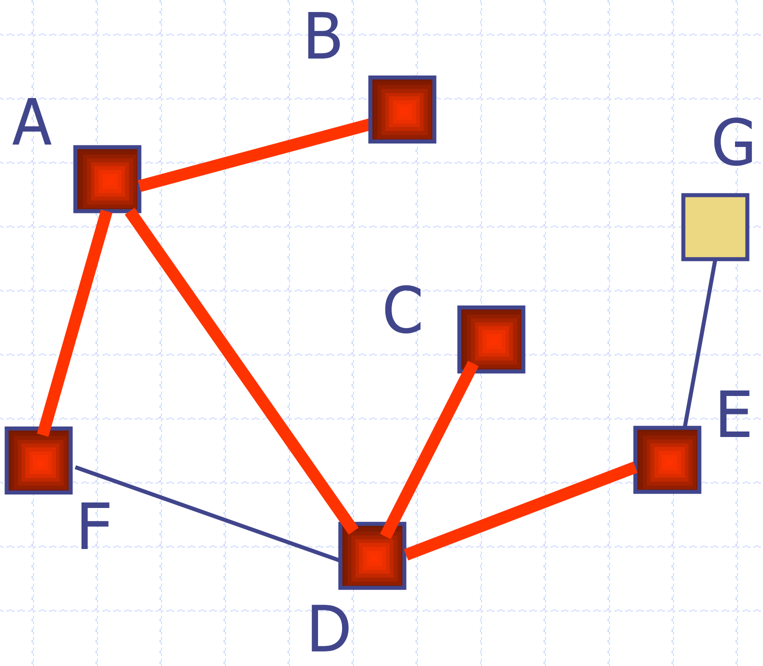
Spanning Tree: T = {AB, AD, AF, ...}

Worked Example

Dequeue (removes F); D is adjacent to F but already visited, so we don't visit D again (avoids cycle creation) Dequeue (removes D); C and E are adjacent to D and not yet visited, so mark them and add to the queue.



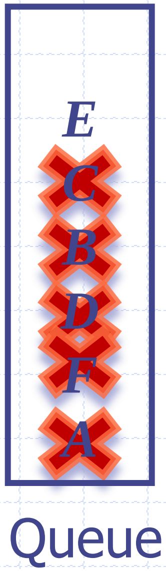
Queue

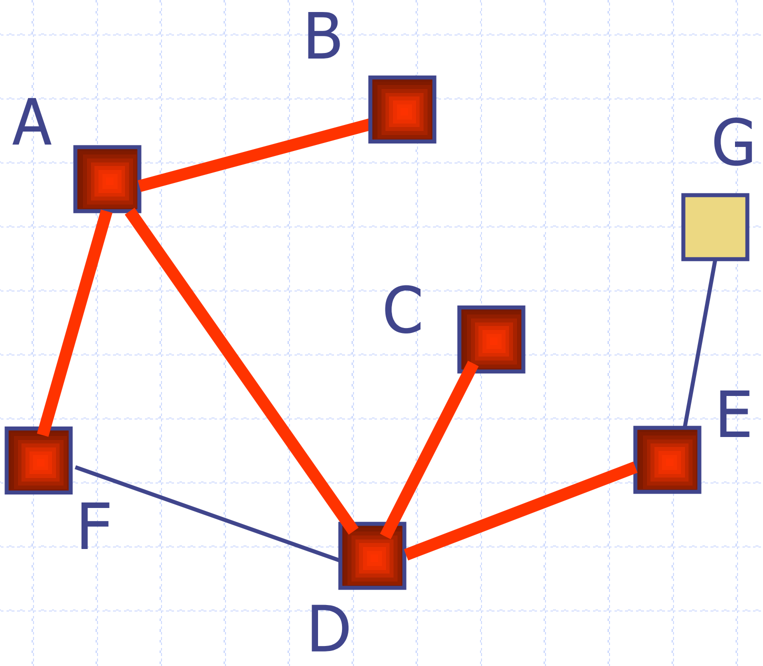


Spanning Tree: T = {AB, AD, AF, DC, DE ...}

Worked Example

Dequeue twice (removing B and C); B and C have no more unvisited adjacent vertices.

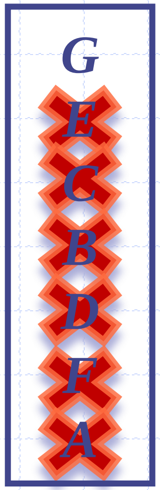




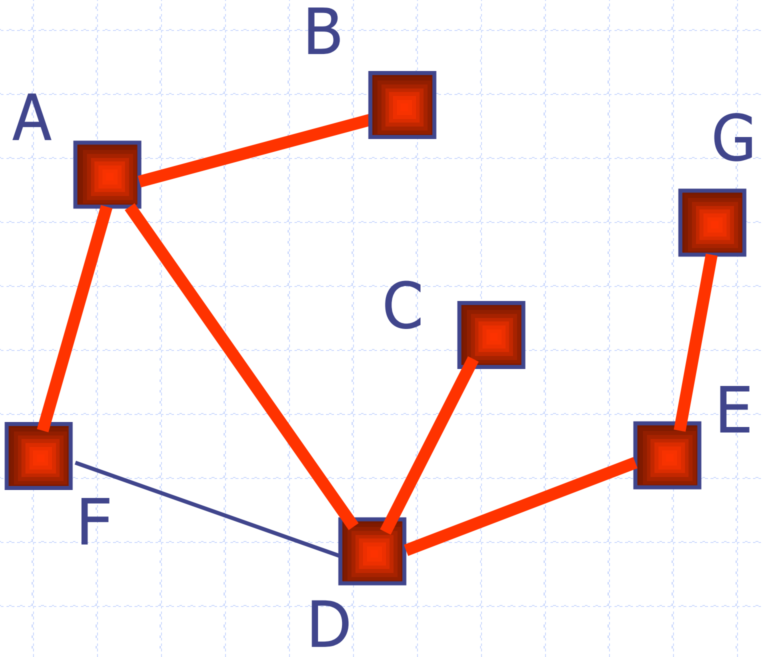
Spanning Tree: T = {AB, AD, AF, DC, DE ...}

Worked Example

Dequeue (removes E); G is adjacent to E and not yet visited. Mark G and add it to queue.



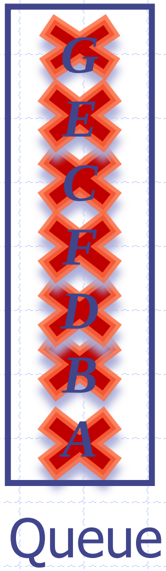
Queue

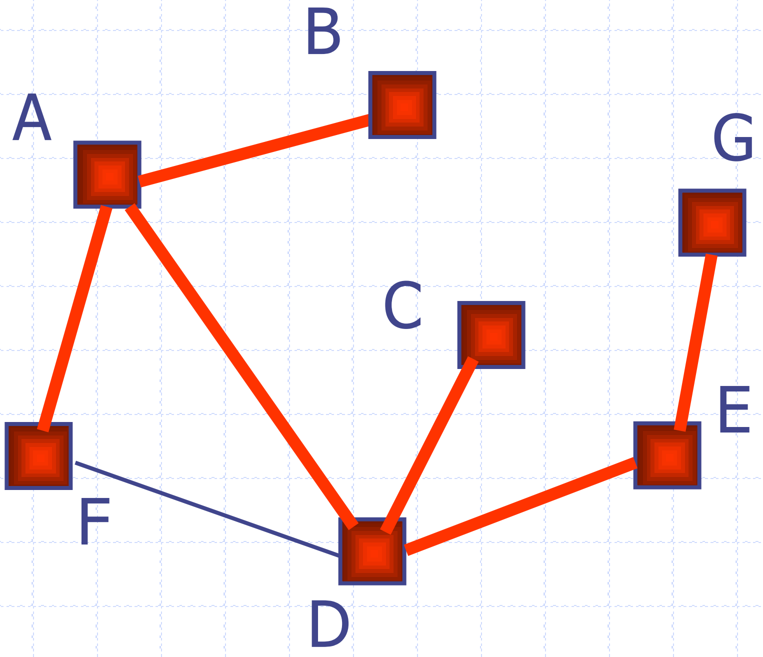


Spanning Tree: T = {AB, AD, AF, DC, DE, EG…}

Worked Example

Dequeue (removes G); The queue is now empty, the algorithm stops.





Spanning Tree: T = {AB, AD, AF, DC, DE, EG}

Running time for BFS

Analysis is similar to that for DFS. In the outer loop, 2 steps of processing occur on each vertex v, and then all edges incident to v are examined (some are discarded, others are processed). Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum is then

$$O(\Sigma_v (1 + deg(v))) = O(n + 2m) = O(n+m)$$

Rework BFS using the template - AbstractGraphSearch

```
Algorithm: AbstractGraphSearch
  while (some vertex not yet visited)
        handleInitialVertext()
        singleComponentLoop()
        additionalProcessing()
```

Rework BFS using the template

- Make BFS be a subclass of AbstractGraphSearch, then we need to take care of the abstract methods from AGS.

```
Algorithm: handleInitialVertex
  Initialize a queue Q
  Pick a vertex s not yet visited
       Q.add(s)
       mark s as visited
       processVertex(s)
```

Rework BFS using the template

Algorithm: SingleComponentLoop

Input: A simple connected undirected graph $G = (V,E)$

Output: G, with all vertices marked as visited.

```
while $Q \neq \emptyset$ do
    v $\leftarrow$ Q.dequeue()
    for each unvisited w adjacent to v do
        mark w as visited
        Q.add(w)
        processEdge(v, w)
        processVertex(w)
```

These two methods again become template methods.

Finding a Shortest Path

- A special characteristic of the BFS style of traversing a graph is that, with very little extra processing, it outputs the shortest path between any two vertices in the graph. (There is no straightforward to do this with DFS.)
- If p: $v_0 - v_1 - v_2 - \dots - v_n$ is a path in G, recall that its length is n, the number of edges in the path. BFS can be used to compute the shortest path between any two vertices of the graph.
- As with DFS, the discovered edges during BFS collectively form a spanning tree (assume G is connected, so there is a spanning tree). A tree obtained in this way, starting with a vertex s (the *starting vertex when performing BFS*) is called the *BFS rooted spanning tree.*

Finding a Shortest Path

- Recall that a rooted tree can be given the usual *levels.*
- Given a connected graph G with vertices s and v, Here is the algorithm to return the shortest length of a path in G from s to v
  - Perform BFS on G starting with s to obtain the BFS rooted tree T with root s, together with a map recording the levels of T
  - Return the level of v in T
- Note: Please refer to the lecture notes to see why this algorithm works.

Finding a Shortest Path

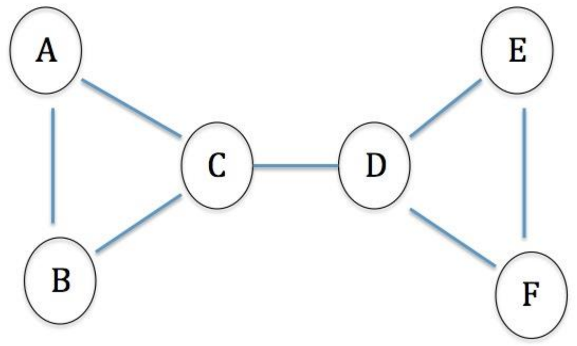
- *Algorithm for computing shortest path* from s to any vertex v:
  - Above, we described an algorithm for obtaining shortest *path length* from s to a vertex v – simply read off the level of v. Here we compute a *shortest path* from s to v
    - Perform BFS, beginning with s. Keep track of the levels of vertices as the spanning tree for this component is being built. Also, keep a hashtable associating with each vertex w a shortest path P[w] from s. Initially, P[s] = ∅. When considering edge (v,w), if w is unvisited, set P[w] = P[v] U (v,w).

Main Point

The BFS and DFS algorithms are procedures for visiting every vertex in a graph. BFS proceeds “horizontally”, examining every vertex adjacent to the current vertex before going deeper into the graph. DFS proceeds “vertically”, following the deepest possible path from the starting point. These approaches to graph traversal are analogous to the horizontal and vertical means of gaining knowledge, as described in SCI: The horizontal approach focuses on a breadth of connections at a more superficial level, and reaches deeper levels of knowledge more slowly. The vertical approach dives deeply to the source right from the beginning; having fathomed the depths, subsequent gain of knowledge in the horizontal direction enjoys the influence of the depths of knowledge already gained.

Special Uses of DFS and BFS

- Both DFS and BFS can be used to compute connected components and a spanning tree
- BFS can be used to compute shortest paths.
- (Optional) DFS can be used to compute the *biconnected components.*



G is *biconnected* if removal of an edge does not disconnect G.

A *biconnected component* is either a separating edge or is a subgraph that is maximal with respect to being biconnected

Biconnected components: A-B-C, C-D, D-E-F

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Both DFS and BFS algorithm provide efficient procedures for traversing all vertices in a given graph.
2. By tracking edges during DFS or BFS, we can obtain a spanning tree/forest for a given graph without much effort.
3. Transcendental Consciousness is the field of all possibilities, located at the source of thought by an effortless procedure of transcending.
4. Impulses within the Transcendental field: The entire structure of the universe is designed in seed form within the transcendental field, all in an effortless manner.
5. Wholeness moving within itself: In Unity Consciousness, each expression of the universe is seen as the effortless creation of one's own unbounded nature.